

ACCELERATING COVERAGE CLOSURE FOR HARDWARE VERIFICATION
USING MACHINE LEARNING

A Thesis

by

AMRUTHA SHIKARIPURA JAGADEESH

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Jiang Hu
Co-Chair of Committee,	Aakash Tyagi
Committee Member,	I-Hong Hou
Head of Department,	Miroslav M. Begovic

May 2019

Major Subject: Computer Engineering

Copyright 2019 Amrutha Shikaripura Jagadeesh

ABSTRACT

Functional verification is used to confirm that the logic of a design meets its specification. The most commonly used method for verifying complex designs is simulation-based verification. The quality of simulation-based verification is based on the quality and diversity of the tests that are simulated. However, it is time consuming and compute intensive on account of the fact that a large volume of tests must be simulated to exhaustively exercise the design functionality in order to find and fix logic bugs. A common measure of success of this exercise is in the form of a metric known as functional coverage. Coverage is typically indicated as a percentage of functionality covered by the test suite. This thesis proposes a novel methodology to construct a model using SVM, Gradient Boosting Classifier and Neural Networks aimed at replacing random test generation for speeding up coverage collection.

ACKNOWLEDGEMENTS

I would like to thank Dr. Jiang Hu and Dr. Aakash Tyagi for their continuous support and immense knowledge. My heartiest gratitude to them to have allowed me to work with them and extend their ideas on this topic.

I would like to thank Professor Michael Quinn for sharing his immense knowledge and experience in giving us inputs to help go forward with our ideas. I would also like to thank Dr. I-Hong Hou, for willing to be a part of my thesis committee and review panel and providing thoughtful insights.

Finally, I would like to thank my family and friends for their encouragement and moral support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Professors Jiang Hu and I-Hong Hou of the Department of Electrical and Computer Engineering and Professor Aakash Tyagi of the Department of Computer Science and Engineering.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This work did not receive any external funding.

TABLE OF CONTENTS

	Page
ABSTRACT	ii
ACKNOWLEDGEMENTS	iii
CONTRIBUTORS AND FUNDING SOURCES.....	iv
TABLE OF CONTENTS	v
LIST OF FIGURES.....	vii
LIST OF TABLES	ix
1. INTRODUCTION.....	1
1.1. Overview	1
1.2. Background on Verification.....	1
1.2.1. Code coverage	2
1.2.2. Functional coverage	2
1.2.3. Constrained Random Verification.....	3
1.3. Motivation	4
1.4. Background in Machine Learning.....	5
1.4.1. Neural Networks.....	6
1.4.2. Support Vector Machine	7
1.4.3. Gradient Boosting Classifier	8
2. PRIOR WORK.....	11
3. DESIGN UNDER TEST.....	13
3.1. Pseudo-Least Recently Used (LRU) Algorithm.....	13
3.2. MESI Protocol.....	14
4. ACCELERATING COVERAGE CLOSURE FOR HARDWARE VERIFICATION USING MACHINE LEARNING.....	16
4.1. Overview	16
4.2. Methodology	16
4.3. Basic Blocks	18

4.3.1. Random Test Generation Block	18
4.3.2. Coverage Collection Block	19
4.3.3. New Coverage Collection Block	20
4.3.4. Machine Learning Model	20
4.4. The overall flow of the model	20
4.5. Coverage Data	21
5. RESULTS.....	23
5.1. Model Constructed Based on Entire Coverage Data.....	23
5.2. Model Constructed Based on Partial Coverage Data	26
5.3. Separated Models Based on Individual Coverpoints	30
6. CONCLUSION	37
REFERENCES	38

LIST OF FIGURES

	Page
Figure 1-1 Graph showing the correlation between Failure Rate and Coverage Progress.....	5
Figure 1-2 On the right is a graph that shows linear classification for a linear model. On the right is an example where the initial non-linear data plotted in XY plane is transformed to XZ plane using kernel function.....	8
Figure 1-3 Visualizing the gradient boosting model adaptation for the first four iterations.	10
Figure 1-4 Visualizing the gradient boosting model adaptation at the 18 th and 19 th iteration.	10
Figure 3-1 MESI coherency protocol state diagram	15
Figure 4-1 Basic block diagram of the model	18
Figure 5-1 On the left is the percentage coverage curve v/s the number of tests for conventional random tests. On the right is the Mean Squared Error curve v/s the number of tests for the SVM model with 280 input features.....	24
Figure 5-2 Graph comparing the performance of the three models with random test generation for 12 coverpoints.	26
Figure 5-3 Coverage plots of a few coverpoints which reach 100% within 20 tests	28
Figure 5-4 Coverage plot after excluding eight coverpoints, which has 280 coverbins. .	29
Figure 5-5 Coverage curves for 122 coverbinss.....	29
Figure 5-6 Coverage curves comparing all three models with the random test generation results for X_PROC__REQ_TYPE coverpoint, which has 16 coverbins.	33
Figure 5-7 Coverage curves comparing all three models with the random test generation results for X_PROC__DATA coverpoint, which has 80 coverbins.	33
Figure 5-8 Coverage curves comparing all three models with the random test generation results for X_PROC__SNOOP coverpoint, which has 16 coverbins.	34

Figure 5-9 Coverage curves comparing all three models with the random test generation results for X_PROC__SNOOP_WR coverpoint, which has 10 coverbins.	34
-------------------------------------------------------------------------------------------------------------------------------------------------------------	----

Figure 5-10 Coverage performance curves for each model having combined the model results of four coverpoint, which totally has 122 coverbins.	35
----------------------------------------------------------------------------------------------------------------------------------------------------	----

LIST OF TABLES

	Page
Table 3-1 Pseudo-LRU replacement policy	14
Table 4-1 Coverpoints and their respective coverbins for the considered design.....	22
Table 5-1 List of training and prediction test count to get 100% coverage (280 coverbins), along with the percentage improvement when compared to results with no model applied.	25
Table 5-2 Table comparing overall simulation time	26
Table 5-3 Comparing the test count of the three models with the random test generation result for 122 coverbins.	30
Table 5-4 Table listing the training and prediction time for all four coverpoints. The time listed is in seconds.	31
Table 5-5 Table listing the training and prediction test count for all four coverpoints....	32
Table 5-6 Time and test count improvement for all three model construction methods..	36

1. INTRODUCTION

1.1. Overview

Complex microprocessor designs rely heavily on simulation-based verification to effectively verify the design. Over the years, the complexity of designs has increased drastically manifested by an increase in the die area and the transistor density, and an increase in the time required to verify the designs. Roughly 70% of the design effort goes into verification. Machine learning has been successfully applied for various applications such as virtual personal assistance, video surveillance, social media services, online customer service and many more. The research work presented in this thesis is an attempt to adopt machine learning techniques to speed up design verification. Specifically, our work aims to provide a smarter test generation model to reduce simulation time. The constrained random test generation method is modulated with the coverage data. SystemVerilog testbench with Universal Verification Methodology (UVM) framework is integrated with the machines learning models.

1.2. Background on Verification

Simulation-based verification has three important components: test generation, checkers and coverage. Randomly generated tests make use of the test templates written by verification engineers and the constraints are applied to the test parameters. This is known as constrained random test generation. Checkers are added to monitor the functional correctness of the design under test (DUT). Coverage is a metric used to measure the effectiveness of the simulation on the DUT. A satisfactory level of coverage

should be achieved to tape out the design. Conventionally, this analysis is done in a manual fashion thereby adding to the verification time. Coverage can be divided into code coverage and functional coverage.

1.2.1. Code coverage

Code coverage is the implementation coverage. It indicates a measure of exercise of any given line of the hardware description language (HDL) code by the test. This coverage is useful in the initial phase of verification. Code coverage is of the following types, block coverage, toggle coverage, and expression coverage. A 100% code coverage does not mean that the DUT is completely verified since line (code) coverage does not embody the expression of underlying functionality unless it is seen in the full context of the design functionality typically assembled as a combination of multiple lines of code. It cannot ensure the quality of verification and its completeness.

1.2.2. Functional coverage

Functional coverage is the specification coverage. It checks if the design implementation meets the design specification by examining functional correctness. With proper functional metrics defined, it can be made to cover all features mentioned in the specification. It can be made to check for all possible range of values, and design boundaries and limitations. For this reason, we can rely on functional coverage more than code coverage as a measure of functionality.

Functional coverage in SystemVerilog is defined using the covergroup construct in which we define coverpoints and coverbins. The below example defines a covergroup for a memory block. The term addr is a coverpoint for address that has values ranging

between 0 and 255. Coverbins are defined to split these values into multiple small ranges known as bins. Read_write coverpoint has only two possible values, 0 and 1 which are this coverpoint's coverbins.

```
Covergroup memory @(posedge clk);    //defining a covergroup
    address : coverpoint addr {      // defining a coverpoint for address
        bins low      = {0,50};      // defining the coverbins for address
        bins medium = {51,150};
        bins high     = {151,255};
    }
    read_write : coverpoint rw {      // defining read write coverpoint
        bins read  = {0};             // defining the coverbins for read
                                         // and write
        bins write = {1};
    }
endgroup : memory                  // end of the covergroup
```

A coverpoint is considered completely covered only if each of its bins is covered at least once. Therefore, with the increasing complexity of the design, the conventional approach for meeting the 100% coverage goal becomes time-consuming.

1.2.3. Constrained Random Verification

Constrained random verification has random test generation that is better than the directed test since it shifts the burden of comprehensive testcase generation from the verification engineer to the machine. The constraints restrict the generation to valid tests, whose values can be chosen by the designer. Many tests with a random combination of

constraints give higher chances of covering corner cases as compared to directed test cases.

```
rand len;                // random test parameter declared as a rand variable.  
constraint legal_lengths {  
    len >= 2;            // Constraints for the variable.  
    len <= 13;  
}
```

In the above example, the term len is defined as a random variable that is constrained to a certain set of values. The randomize() method is later used to generate a random value within the range that is defined by the constraint. The essential steps in coverage driven verification can be stated as follows,

1. Set up coverage model made up of covergroups and coverpoints.
2. Set up checkers.
3. Debug the verification environment.
4. Perform random tests generation and collect coverage.
5. Update constraints to target the cover holes and run more tests.
6. Analyze and run directed tests to cover holes until the targeted coverage is reached.

1.3. Motivation

Design verification plays an important role in the design flow by ensuring the correctness of the design. As we approach the knee of the coverage curve, the effort required to gain incremental coverage becomes increasingly difficult. Figure 1-1 shows

the correlation between the failure rate and coverage progress. Here we see that the it takes a considerable amount of time to reach 100% coverage. The work presented in this thesis aims at shifting the coverage curve leftward and upward after the failure rate is considerably low, effectively accelerating the coverage collection process.

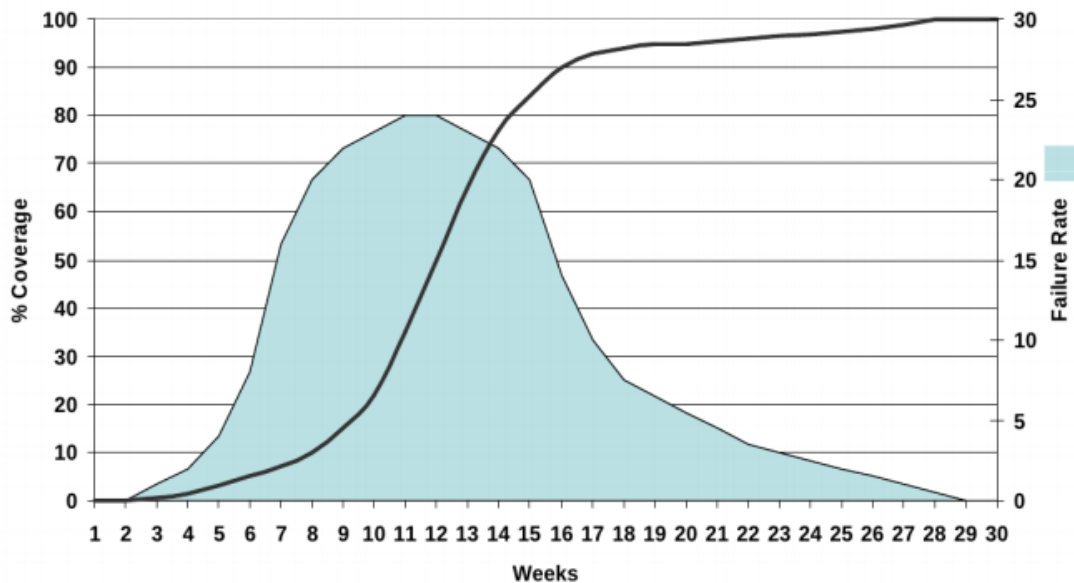


Figure 1-1 Graph showing the correlation between Failure Rate and Coverage Progress

1.4. Background in Machine Learning

Machine learning is used to automate the analysis of data by building an analytical model. It applies learning algorithms enabled with computational power of computers to solve problems with big data. Most fields that work with large data have recognized the importance of machine learning. These algorithms are broadly

categorized into supervised and unsupervised learnings. Supervised learning requires the user to provide both the input and the respective output values to train the model. Here the input is known as an input feature and output as a label. Unsupervised algorithms do not need an output to train the model. They instead use an iterative method for learning.

1.4.1. Neural Networks

A neural network is statistical regression implemented in a brain-like structure. A neural network contains three basic layers: an input layer, a hidden layer and an output layer. Input to the network is fed through the input layer. A hidden layer processes the data fed through the input layer that is made available by the output layer. A network can have one or more hidden layers. Neuron is the basic building block of a neural network. An activation function is applied to the weighted sum of the inputs to a neuron to obtain the output. If a neuron has $n+1$ inputs, X_0 to X_n with weights W_0 to W_n and with an activation function of f , then the output Y of the neuron is given by,

$$Y = f \left(\sum_{i=0}^n X_i * W_i \right)$$

Activation function is of two types: linear and non-linear. In linear activation function, the output is linearly related to the input. The value is not limited to any value like in non-linear activation function. Some of the non-linear activation functions include binary step, sigmoidal or logistic, rectified linear unit (ReLU), and Tanh.

We train the model with a given number of training data set in terms of epochs. Each epoch is the number of times the algorithm sees all samples in the data set. The lesser the number of epochs, the lower the precision of the model. But a high epoch count can lead to overfitting of the model. Thus, to ensure an optimal model, the number

of epochs should be carefully chosen. Back-propagation algorithm is often employed to update the weights used in the network. The training begins with randomly assigned weights. Prediction error is calculated for each iteration and is used to re-evaluate and update the weights.

1.4.2. Support Vector Machine

A Support Vector Machine (SVM) is a classifier defined by a separating hyperplane. In supervised learning, given labeled training data, the algorithm outputs an optimal hyperplane that is used to categorize data. This hyperplane divides the plane such that each class lays on either side. This algorithm aims at finding the margin, which is the largest minimum distance to the training data. An optimal separating hyperplane maximizes the margin of the training data. Finding the optimal hyperplane is simple when the data points are linearly separable. For non-linear separation of data, a kernel trick is used. The kernel is a set of functions that transform the input space from a lower dimension to a higher dimension to ease classification. Figure 1-2 [11] gives an example of both linear and non-linear classification. The figure on the left is a linear classifier with an optimal plane and maximum margin for the given test data. On the right are two figures that show the transformation from a lower dimension to a higher dimension. The training data is represented in its original dimension in the XY plane. By transforming it to a higher dimension, which is the XZ plane, the non-linear classification is transformed into a linear classification.

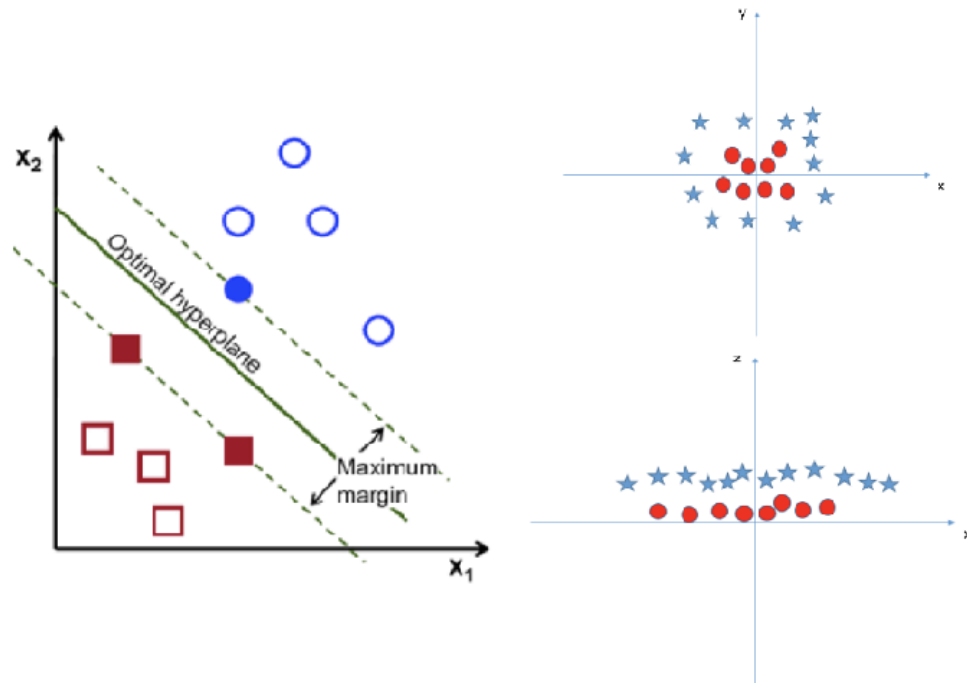


Figure 1-2 On the right is a graph that shows linear classification for a linear model. On the right is an example where the initial non-linear data plotted in XY plane is transformed to XZ plane using kernel function.

1.4.3. Gradient Boosting Classifier

An ensemble is a collection of predictors that are used to give a final prediction. The reason we use ensembles is that many different predictors together will perform better than any single predictor alone. This technique can be classified into bagging and boosting technique based on how the predictors are integrated. Bagging technique involves building many independent predictors and combining them using some model averaging techniques. In boosting technique, the predictors are made sequentially, where subsequent predictors learn from the mistakes of the previous predictors. It takes fewer iterations to reach closer to actual predictions because new predictors learn from mistakes committed by previous predictors. The methods of [7] and [8] helps us

understand how boosting can be used as a machine learning approach and also learn about the non-linear classifiers. The predictors can be chosen from a range of models like decision trees, regressors, classifiers, etc.

Gradient Boosting is an example of the boosting method. It is a learning technique for regression and classification problems whose objective is to define a loss function and minimize it. Figures 1-3 and 1-4 [12] help us understand how the model gets better by assessing the error of the previous iteration. In the figures, the blue dots show the input vs output curve and the red line shows the values predicted by the model. Residual for a model is the difference of the actual value and the predicted value. This is a measure of error in the model prediction. The green dots show the residual versus the input curve for an iteration. Here the iteration represents the sequential order of fitting the gradient boosting algorithm. In short, we first model data with simple models and analyze data for errors. Using these errors, we identify the data points that are hard to fit, which are used by later models to get them right. In the end, all predictors are combined with each predictor having some weight.

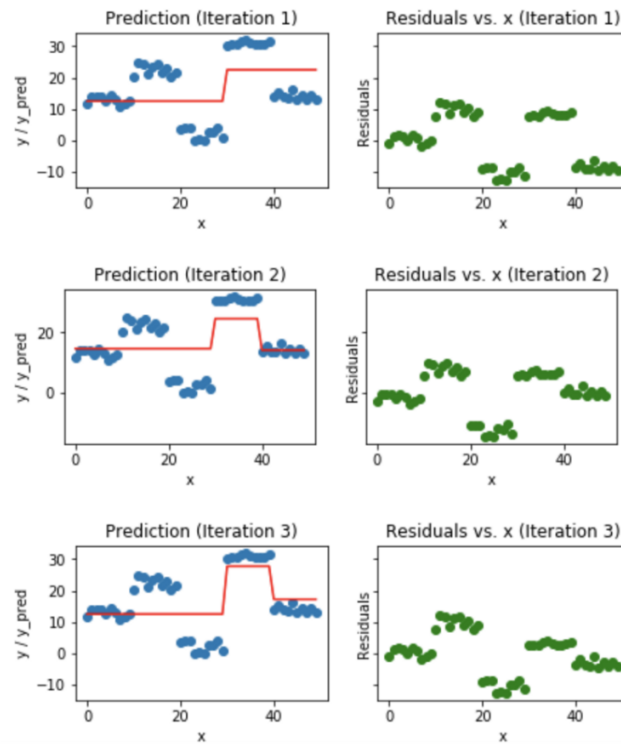


Figure 1-3 Visualizing the gradient boosting model adaptation for the first four iterations.

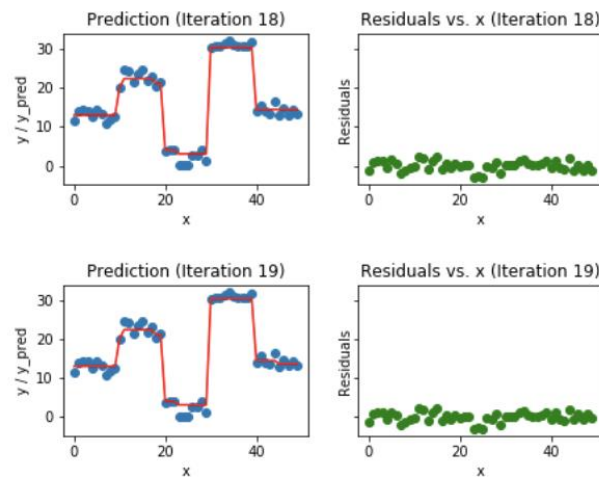


Figure 1-4 Visualizing the gradient boosting model adaptation at the 18th and 19th iteration.

2. PRIOR WORK

In recent years there has been a considerable amount of effort in applying machine learning algorithms in the field of verification. Applying them in coverage improvement is one of the common applications. A methodology is proposed in [1], where toggle coverage metrics and machine learning algorithms are utilized to create tests with higher potential of exposing previously uncovered regions in the design. Each test is given a score to measure its stress factor. Unfortunately, toggle coverage is not a very effective metric to capture design functionality. Bayesian network is implemented in [2] to perform Coverage Directed test generation. The Bayesian network is trained with the data from the multiple test runs and the coverage collected. This trained network is then used to generate tests that help to cover all possible transactions from the CPU in their design. The method proposed in [3] presents feature-based rule learning to extract knowledge that is used to improve assertion-based coverage. Test generation based on coverage using Unsupervised Support Vector Analysis (SVA) is proposed in [4]. The model obtained is used to exercise the previously unexercised functionality. For this, it makes use of constrained random test generation along with kernel-based support vector analysis that works on test selection. The method outlined in [5] make use of Supervised SVA along with coverage-driven test generation to help generate more directed tests that help improve coverage performance and reduce human effort. Test generation is optimized by pruning tests, which uses SVA. In [6] Artificial Neural Network (ANN) is used to identify the critical test features and the important

coverpoints. This extracted knowledge is later used to guide the coverage-driven test generation to accelerate the verification process.

3. DESIGN UNDER TEST

The design under test is a four-core design. Each core has its own Level 1 cache, and a shared Level 2 cache. Each Level 1 cache is split into instruction level cache and data level cache. Both instruction and data level caches are 256 KB 4-way set associative, with the pseudo-LRU replacement policy, and Modified Exclusive Shared Invalidate (MESI) based coherency protocol. Data level and instruction level cache share one bus to communicate with the core.

3.1. Pseudo-Least Recently Used (LRU) Algorithm

Cache replacement algorithms are used for improving the performance of the cache by making the optimal choice for replacement when the cache is full. Least recently used algorithm keeps a record of the most often used or recently used and replaces the least frequently used memory when the cache is filled. To do so, it keeps track of the state of each memory block in cache using age bits. Value of the age bits is directly proportional to the frequency of its use. One disadvantage is that as the number of slots in a cache that needs keeping track, even the number of age bits required increases. Pseudo-LRU algorithm works like an LRU algorithm but has a fewer bits for representing age. Table 2-1 indicates the state replacement and next state transition relation for implementing pseudo-LRU in a 4-way set associative cache. The 'x' indicates don't care and '-' means unchanged.

Table 3-1 Pseudo-LRU replacement policy

State replacement relation		Next state transition	
State	Replacement	Refer to	Next state
00x	Line 0	Line 0	11-
01x	Line 1	Line 1	10-
1x0	Line 2	Line 2	0-1
1x1	Line 3	Line 3	0-0

3.2. MESI Protocol

Designs with multiple core and each core having local cache means multiple copies of a memory block is available simultaneously. To ensure that the correct value of the memory block is used, we apply coherency policy. MESI coherency protocol is used, which has Modified, Exclusive, Shared and, Invalid are the main states. All possible states and the criteria for its transition is given in Figure 2-1, which is taken from [13].

- **Modified State** – Cache line is present in the current cache only and is modified. The cache is required to be written back to main memory for future transactions which involves other caches requesting this cache line.
- **Exclusive State** - The cache line is present only in the current cache but is clean - it matches main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when writing to it.

- Shared State - Indicates that this cache line is present in more than one cache and is clean - it matches the main memory.
- Invalid State - Indicates that this cache line is invalid (unused).

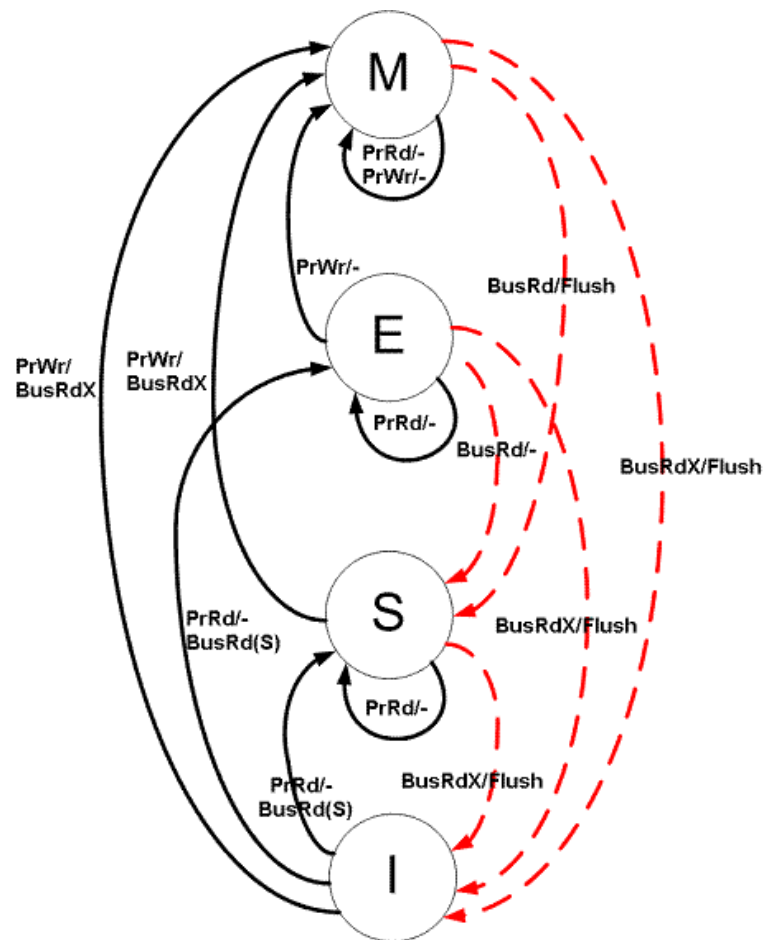


Figure 3-1 MESI coherency protocol state diagram

4. ACCELERATING COVERAGE CLOSURE FOR HARDWARE VERIFICATION USING MACHINE LEARNING

4.1. Overview

In the previous works, we have seen methods for test pruning, important feature selection for tests, identifying important coverage and assertion properties. During test pruning, the input features are the tests and output labels are coverage data. Among the randomly generated tests, those that are likely to improve the coverage are chosen. In our work, we use coverage data as input feature and test as an output label. On training the model, we input the exact coverage that is needed, and the model generates the test. The machine learning algorithms are implemented in Python and integrated with the UVM environment. Keras [9] and Scikit-learn [10] libraries were used for implementing the machine learning algorithms.

4.2. Methodology

To understand the methodology, we need to understand the following two steps: generating input features and output labels, and training the model and using it for prediction.

1. The features and the labels to the Machine Learning Model are the coverage data and the test parameters, respectively. A set of parameters are defined, which can be used to control the constrained random test generation. Tests are controlled by choosing appropriate values for these parameters. These parameters serve as the output labels for the model.

The coverbins collected from the coverage data serve as input features for the model.

2. Random tests are run on the DUT and the coverage data is collected. The collected data is used to train the neural network. Once the training is completed, it is used to predict tests that can uncover coverage holes. This is done by the random generation of possible and legal combinations of coverage holes which are fed as input to the model. The predicted labels are used to generate the required tests, new coverage is collected, and the process repeats.

Figure 4-1 shows the block diagram for the above-mentioned methodology. Here arrows labeled with T show the flow of operations during training. The random test generation block randomly generates the test parameters, which is the same as constrained random generation. These parameters along with the coverage data collected are used to train the model. In the prediction phase, shown by the arrows labeled with P, random legal combinations of the coverage holes are fed to the Neural Network model. These new possible coverage combinations are generated in the New Coverage Generation block. The prediction of the model, which are the test parameters, are directly fed to the DUT, where they are converted to tests using predefined test templates. The new coverage and the merged coverage are monitored.

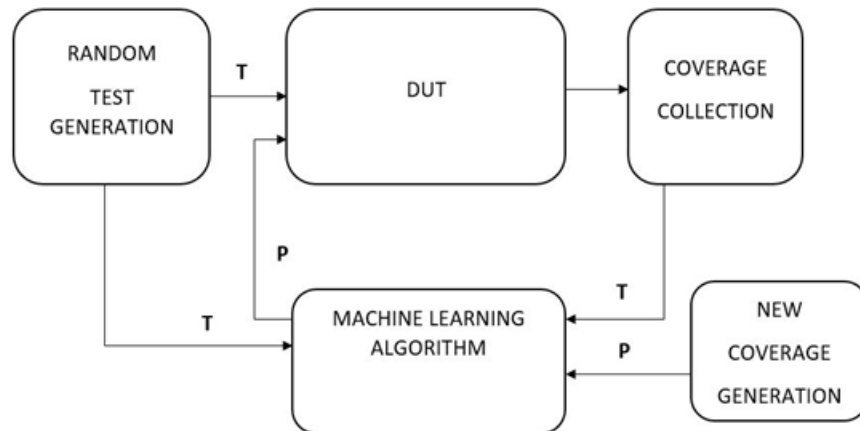


Figure 4-1 Basic block diagram of the model

4.3. Basic Blocks

4.3.1. Random Test Generation Block

This block is used to generate random tests in the training phase. The parameters generated are written in a SystemVerilog test format and is fed to the DUT for simulation and sent to the Machine Learning Algorithm block since these parameters are used as the output label for the model. The actual SystemVerilog tests are controlled by a set of test parameters. These parameters are defined for two reasons. First, we can control the tests better from an outside Python code easily instead of generating the SystemVerilog test. Second, tests can be better represented as labels by doing so.

Each core has Core, Read, Write, Icache and Dcache as test parameters. The Core parameter is used to select which core the test will be directed to, and other parameters control each core's request type and area of the cache. Fix_address and

Parallel_sequence parameters control all four cores. Fix_address is to fix the address for all the read and write requests that are sent to one or more cores. Parallel_sequence parameter is to control the order of execution of requests to cores, either in parallel or in sequence.

4.3.2. Coverage Collection Block

The Coverage Collection block collects the coverage data from the DUT and stores it for future purposes. The collected coverage data is stored in a table in terms of coverpoints and coverbins. Each coverbin is represented in terms of a 0 or a 1, which indicate being covered or not, respectively.

It is important to note that all coverbins are legal and explicitly defined. No illegal, ignore or autogenerated bins are considered. The reason for this is that the proposed model is used for coverage closure purposes. This process happens in the regression stage of the Verification Cycle, and before we reach this stage it would be confirmed that majority of the major bugs and corner case bugs have been found and fixed. Since illegal bins are used for triggering assertions to check for bugs, we do not collect coverage in this event of the regression stage. We do not use autogenerated coverbins as its count can change during random test generation based on its coverpoint values.

Once the coverage data is stored in the required format, it is fed to the Machine Learning Model for training purpose during the training phase. Coverage of newly simulated tests is collected and merged with the previous coverage data and fed to the New Coverage Collection block during the prediction phase.

4.3.3. New Coverage Collection Block

Having the coverage table passed on from the Coverage Collection Block, the overall coverage percentage is checked. If a 100% coverage is obtained, then further simulation is not needed so the overall flow of the model is stopped. If a 100% coverage is not yet obtained, the New Coverage Collection block generates valid coverage combination for the uncovered coverbins. By this, we mean that a possible combination of coverbins which we aim to achieve through simulation is generated. This is passed as input features to the trained Machine Learning model to predict the test that can possibly give us this coverage. The predicted test is simulated, and new coverage is collected, and the flow continues.

If Core was a coverpoint with four coverbins, one for each core, it is represented with a binary number of four digits. If the present coverage obtained is [0110], it means that the second and third core coverbins have already been covered. Then [1101], [1000], [0001] are a few possible new coverages we can aim to achieve.

4.3.4. Machine Learning Model

In the proposed work three different algorithms are implemented, namely SVM, Gradient Boosting Classifier and Neural Network. The input feature of the model is the coverbins, and the output features are the parameters.

4.4. The overall flow of the model

The overall flow of the implemented model can be explained using the below-mentioned steps:

1. Simulate random tests, until satisfactory precision for the Machine Learning model is obtained.
2. Predict tests for obtaining new coverbins.
3. After each new test is simulated, the merged coverage is calculated. If it reaches 100%, we stop the flow. If not, we repeat the prediction and simulation cycles.

4.5. Coverage Data

As mentioned earlier, the coverage data is stored in terms of coverpoints and their respective coverbins. For the given design we have considered a total of 12 coverpoints or 280 coverbins which are shown in Table 4-1. The coverpoints beginning with X are cross coverpoints which are a combination of two coverpoints. In such a coverpoint all possible combinations of the two coverpoints are considered. For example, from the table, we see that REQUEST_TYPE and REQUEST_PROCESSOR have 4 bins each. X_PROC__REQ_TYPE, which is a cross coverpoint of REQUEST_TYPE and REQUEST_PROCESSOR has 16 coverbins, which are all possible combinations between 2 set of 4 coverbins each.

Table 4-1 Coverpoints and their respective coverbins for the considered design.

COVERPOINTS	#COVERBINS
REQUEST_TYPE	4
REQUEST_PROCESSOR	4
REQUEST_ADDRESS	20
READ_DATA	20
X_PROC__REQ_TYPE	16
X_PROC__ADDRESS	80
X_PROC__DATA	80
X_PROC__SNOOP	16
X_PROC__SNOOP_WR	10
X_PROC__SHARED	10
X_PROC__EVICT	10
X_PROC__CP_IN_CACHE	10

5. RESULTS

Implementing the above-explained methodology was done in multiple steps. Eventually, the right Machine Learning Algorithm and the number of input features were decided based on the results obtained. Three different ways of utilizing coverpoints for machine learning model construction are evaluated.

5.1. Model Constructed Based on Entire Coverage Data

In this model construction method, the entire coverage data was considered, that is 280 coverbins as input features to Machine Learning model. We started off with the SVM classification algorithm. The reason for this choice is that the data is represented in binary form and classification can be used effectively. Therefore, 22 SVM models were built for the 22 test parameters that are required to generate a test. As mentioned earlier, the number of tests required for training depends on the precision of the trained model. The precision is calculated using the Mean Squared Error (MSE) value. MSE is the mean of the squared value of the differences of the predicted value to the actual value. Figure 5-1 gives the coverage percentage curve and MSE curve, both measures with respect to the number of tests simulated. From observing the two curves we see that if we target to get an accuracy of around 20%, the coverage has already reached 100% by that time with conventional random tests. So, we were forced to consider both the coverage data as well as the accuracy while deciding the number of tests needed for training. The implementation using SVM yields an improvement of -7.65027% in the number of tests as compared to random test generation, which is considered as the

baseline. This means that it took an extra 7.65027% of the total number of tests to obtain 100% coverage.

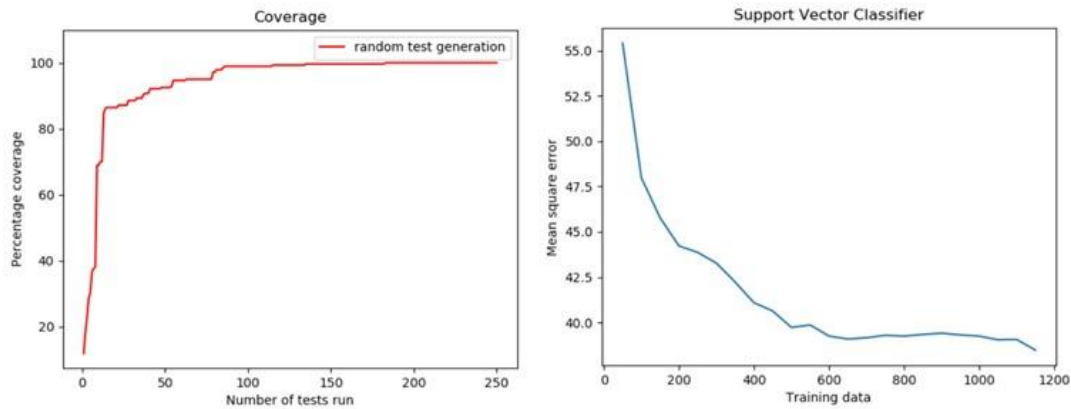


Figure 5-1 On the left is the percentage coverage curve v/s the number of tests for conventional random tests. On the right is the Mean Squared Error curve v/s the number of tests for the SVM model with 280 input features

The reason for the model's failure to yield positive results can be due to the following two reasons. First, the amount of data we are working with could be too much. Second, the machine learning algorithm could be at fault. To confirm the reasoning, we implement other machine learning algorithms with the same data set as well as used for a reduced data set.

When we say that the choice of machine learning engine is poor, it means that the algorithm is not suited for the data. To get a clearer, picture another classification algorithm and a Neural Network was implemented. The other classification algorithm used is Gradient Boosting Classifier. The reason to choose a Neural Network is to check if the output parameter's interdependency affects the results. That is if the interdependency of the 22 test parameters that are predicted affects the overall results.

Since the classifier has a separate model for each of the 22 test parameters, it might not capture the interdependency. A Neural Network was chosen as a single model to give all 22 outputs. The training test count was again chosen after comparing the coverage and MSE curves. The results of the 3 models are compared in Table 5-1 and a graph doing the same is plotted in Figure 5-2.

Table 5-1 List of training and prediction test count to get 100% coverage (280 coverbins), along with the percentage improvement when compared to results with no model applied.

	Tests			
	Training	Prediction	Total	Improvement (%)
No Model			183	
SVM	100	97	197	-7.650273224
GB	100	86	186	-1.639344262
NN	70	92	162	11.47540984

The overall simulation time comparison is given in Table 5-2, where the training time includes the time needed to run the simulations and train the machine learning model with the collected data. The prediction time includes the generation of coverage combination by the New Coverage Generation block and test simulation, collection of new coverage data and repetition the process until we achieve 100% coverage.

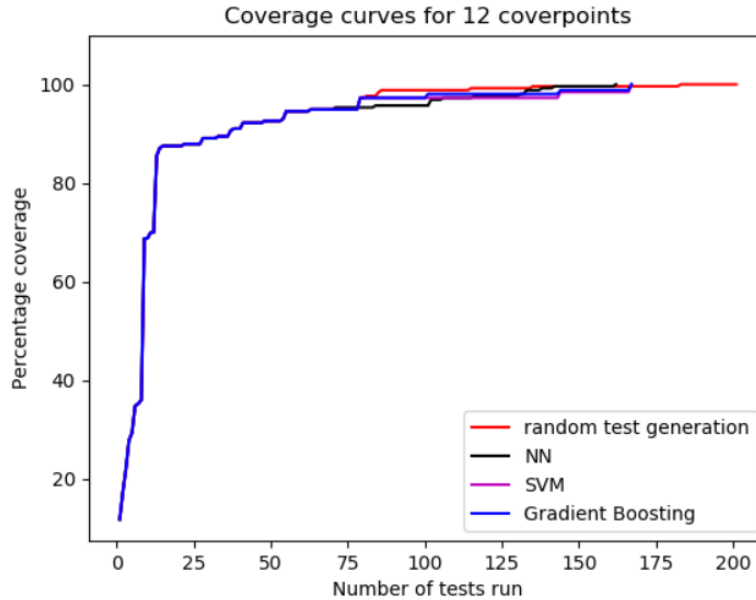


Figure 5-2 Graph comparing the performance of the three models with random test generation for 12 coverpoints.

Table 5-2 Table comparing overall simulation time

	Time (s)			
	Training	Prediction	Total	Improvement (%)
No Model			1492.98	
SVM	735	862.712	1597.712	-7.01496
GB	735	781.41	1516.41	-1.56934
NN	514.5	830.22	1344.72	9.930475

5.2. Model Constructed Based on Partial Coverage Data

In the previous implementation, we dealt with the entire coverage data (280 coverbins). Since the overall performance was not satisfying, we now reduce the number

of coverage data we use to train the model. To decide which ones to consider we examine their coverage curves. If the coverpoints can be easily covered with a few tests, they do not need to be included for machine learning-based acceleration. Out of the coverpoints listed, REQUEST_TYPE, REQUEST_PROCESSOR, REQUEST_ADDRESS, READ_DATA, X_PROC_ADDRESS, X_PROC_SHARED, X_PROC_EVICT and X_PROC_CP_IN_CACHE reach a 100% coverage within 20 tests. Figure 5-3 shows the coverage curve of all the above-mentioned coverpoints.

Excluding the above-mentioned coverpoints, the coverage curve of the remaining four coverpoints is shown in Figure 5-4. We see that when we compare Figures 5-4 and 5-1, the overall coverage curve is not affected by the reduced coverpoints. This is because the tests for the excluded coverpoints and the considered 4 coverpoints are common.

The benefit of this step is that the machine learning model's input feature count reduces from 280 to 122. Again all 3 algorithms were applied, and the results are shown in Figure 5-5 and Table 5-3. From the results, we can see that the performance is comparatively better for all 3 models as compared to the models with 280 coverbins.

From this, we can say that reducing the input features improved the performance. Next, we build a separate model for each coverpoint, further reducing the number of input features to the models, which will be our next model construction method.

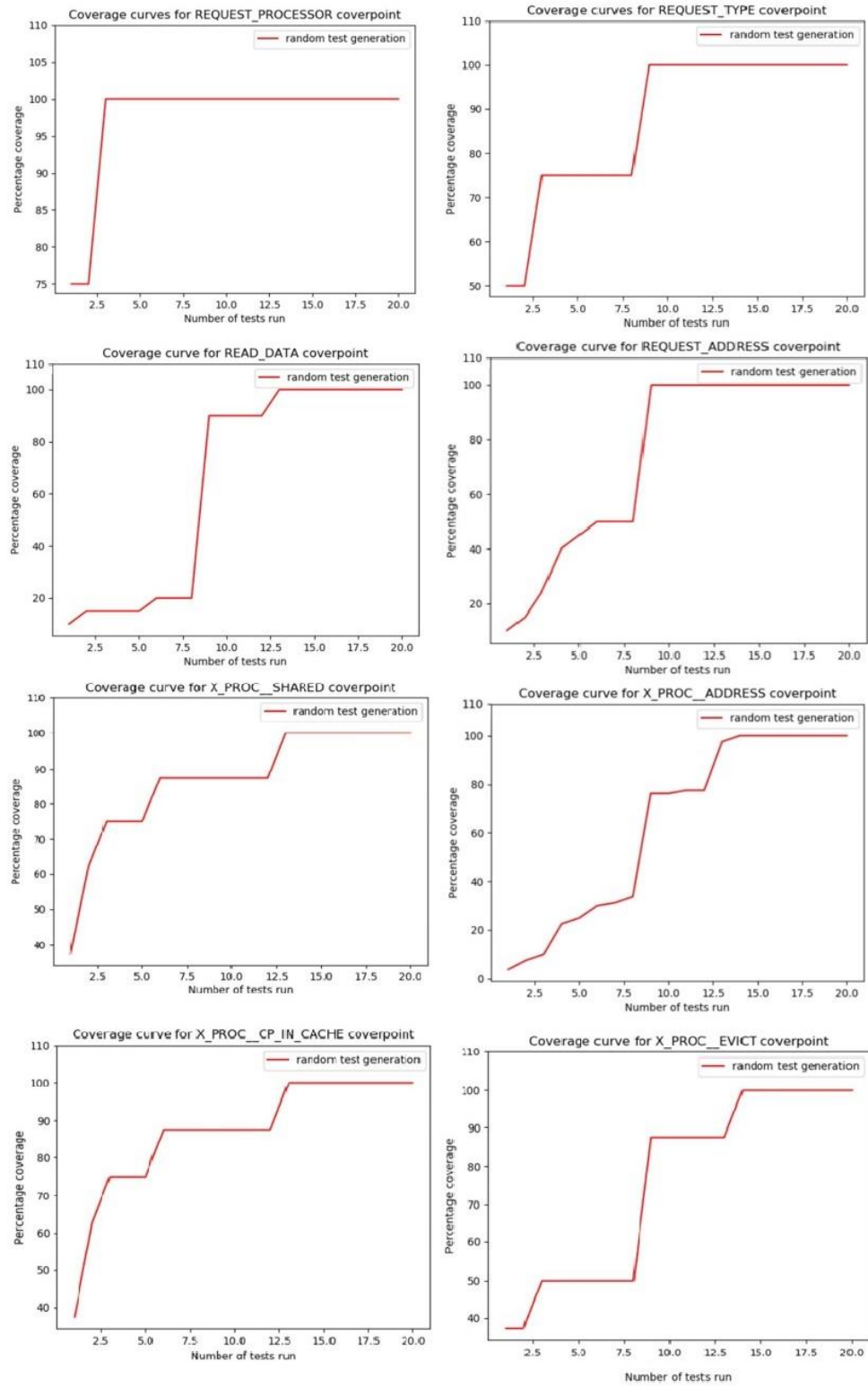


Figure 5-3 Coverage plots of a few coverpoints which reach 100% within 20 tests

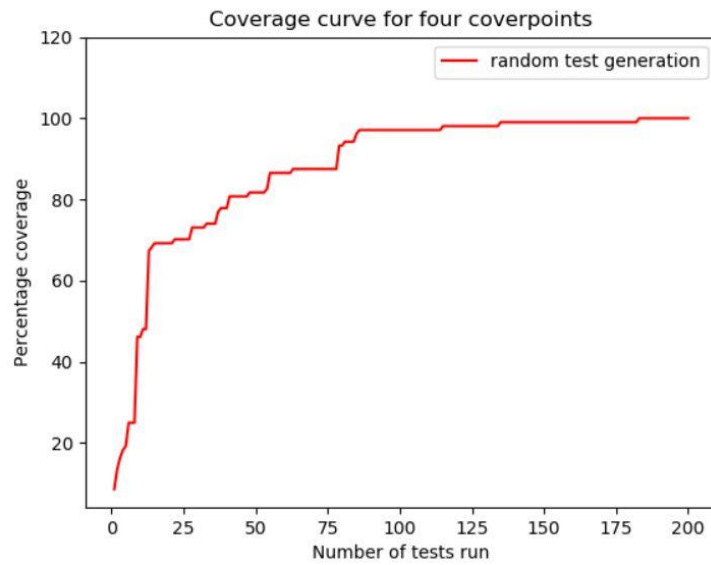


Figure 5-4 Coverage plot after excluding eight coverpoints, which has 280 coverbins.

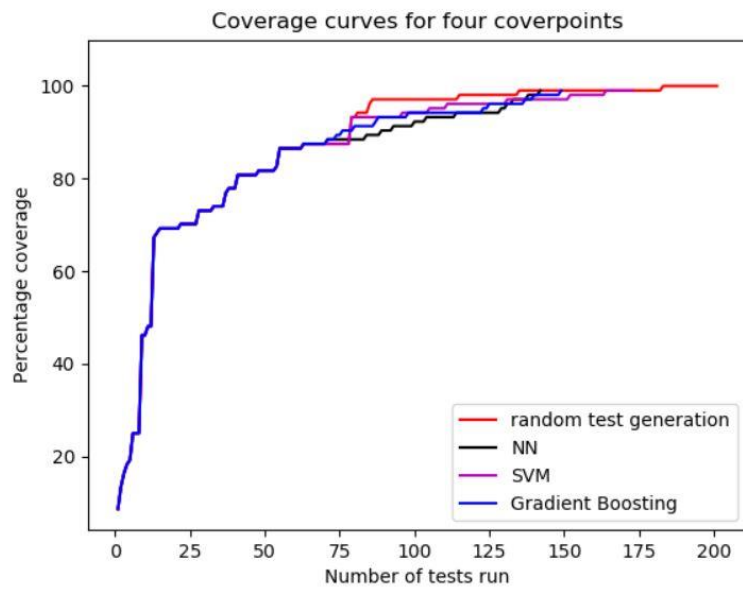


Figure 5-5 Coverage curves for 122 coverbins.

Table 5-3 Comparing the test count of the three models with the random test generation result for 122 coverbins.

	Test			
	Training	Prediction	Total	Improvement (%)
No Model			183	
SVM	80	92	172	6.010929
GB	70	78	148	19.12568
NN	70	71	141	22.95082

5.3. Separated Models Based on Individual Coverpoints

Here a separate model is built for each of the four coverpoints and later merged to check the overall performance. The timing details are given in Table 5-4, while the test count details are given in Table 5-5.

We see that the performance of X_PROC__SNOOP_WR is not impressive. This is because this coverpoint reaches 100% coverage with 24 tests, and we get a lesser window for training our model due to which we must use a model with lower accuracy. If we were to analyze the other coverpoints, we clearly see that Neural Network gives a better result as it captures the interdependency of the coverpoints that the classification models could not capture owing to the separate models for the predicted test parameters. The coverage graphs for the coverpoints are given in Figures 5-6, 5-7, 5-8 and 5-9.

**Table 5-4 Table listing the training and prediction time for all four coverpoints.
The time listed is in seconds.**

X_PROC__REQ_TYPE				
	Training	Prediction	Total	Improvement (%)
No Model			607.88	
SVM	294	22.05	316.05	48.00783049
GB	294	44.1	338.1	44.38046983
NN	220.5	36.75	224.25	63.1094953
X_PROC__DATA				
	Training	Prediction	Total	Improvement (%)
No Model			1387.4	
SVM	882	382.2	1264.2	8.879919273
GB	771.75	316.05	1087.8	21.59434914
NN	551.25	242.55	781.8	43.64999279
X_PROC__SNOOP				
	Training	Prediction	Total	Improvement (%)
No Model			584.72	
SVM	220.5	139.65	360.15	38.40641675
GB	294	110.25	404.25	30.86434533
NN	183.75	58.8	183.75	68.57470242
X_PROC__SNOOP_WR				
	Training	Prediction	Total	Improvement (%)
No Model			208.04	
SVM	133.7	80.85	214.55	-3.129205922
GB	110.25	58.8	169.05	13.45414343
NN	162.15	88.2	230.35	-10.72389925

Table 5-5 Table listing the training and prediction test count for all four coverpoints.

X_PROC_REQ_TYPE				
	Training	Prediction	Total	Improvement (%)
No Model			81	
SVM	40	3	43	46.91358
GB	40	6	46	43.20988
NN	30	5	35	56.79012
X_PROC_DATA				
	Training	Prediction	Total	Improvement (%)
No Model			183	
SVM	120	52	172	6.010929
GB	105	43	148	19.12568
NN	75	33	108	40.98361
X_PROC_SNOOP				
	Training	Prediction	Total	Improvement (%)
No Model			78	
SVM	30	19	49	37.17949
GB	40	15	55	29.48718
NN	25	8	33	57.69231
X_PROC_SNOOP_WR				
	Training	Prediction	Total	Improvement (%)
No Model			24	
SVM	15	11	26	-8.33333
GB	15	8	23	4.166667
NN	15	12	27	-12.5

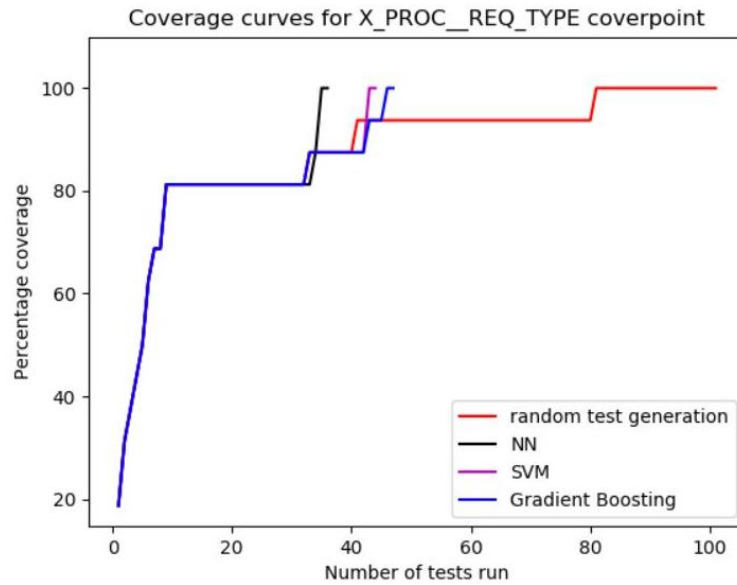


Figure 5-6 Coverage curves comparing all three models with the random test generation results for X_PROC_REQ_TYPE coverpoint, which has 16 coverbins.

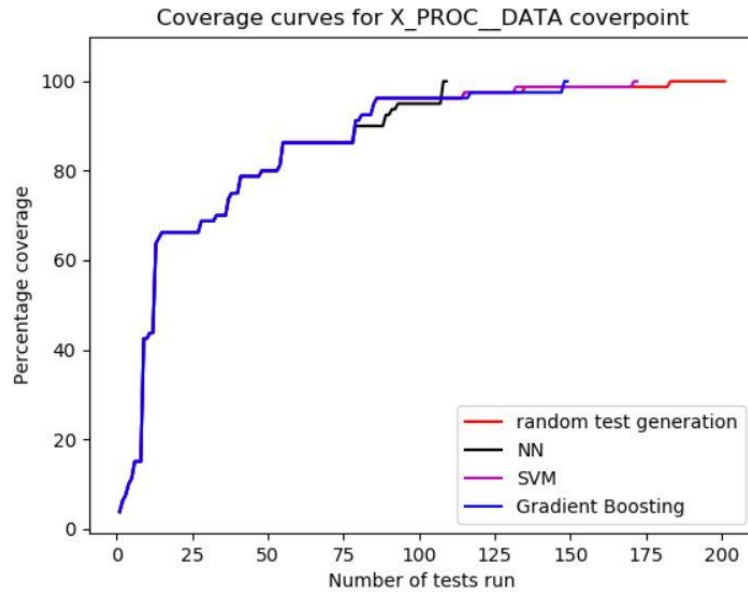


Figure 5-7 Coverage curves comparing all three models with the random test generation results for X_PROC_DATA coverpoint, which has 80 coverbins.

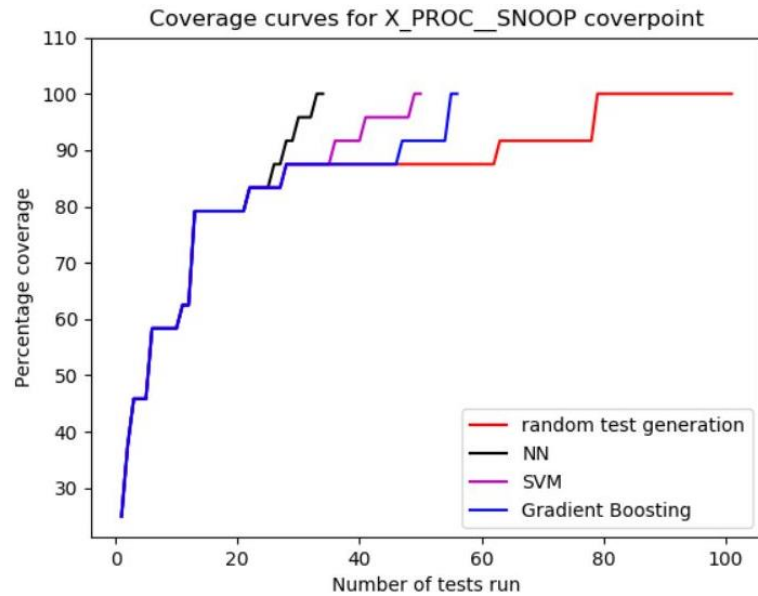


Figure 5-8 Coverage curves comparing all three models with the random test generation results for X_PROC__SNOOP coverpoint, which has 16 coverbins.

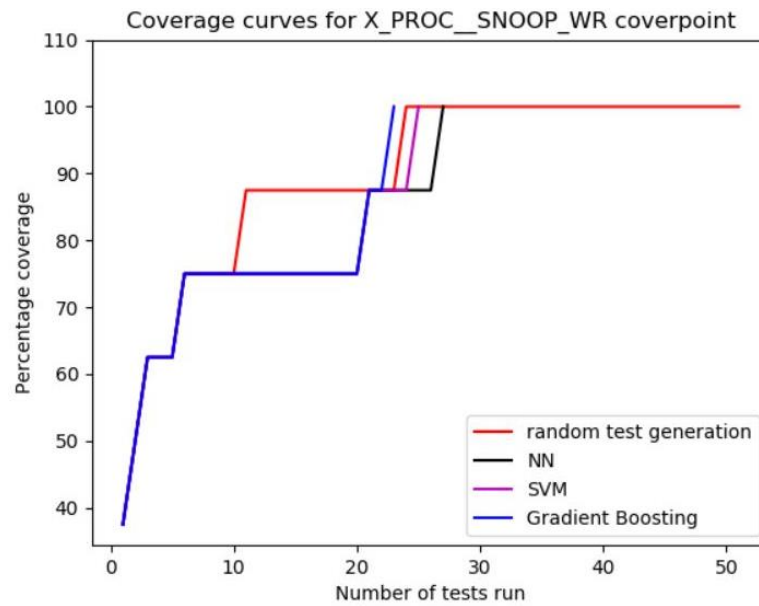


Figure 5-9 Coverage curves comparing all three models with the random test generation results for X_PROC__SNOOP_WR coverpoint, which has 10 coverbins.

When the individual performances of all the models are combined, we get a coverage curve as shown in Figure 5-10. The reason for this is that when we run a test for one coverpoint, then there is a possibility that the same test has affected other coverpoints. After completing one coverpoint, we start training the next model from the present coverage of the coverpoint and not from 0 coverage which helps reduce simulation time. Another advantage is that since we have fewer input features to the model, the training test count is much lesser, and the complexity of the model used is also lesser. Table 5-6 compares the improvement in test count and time of all three implementations.

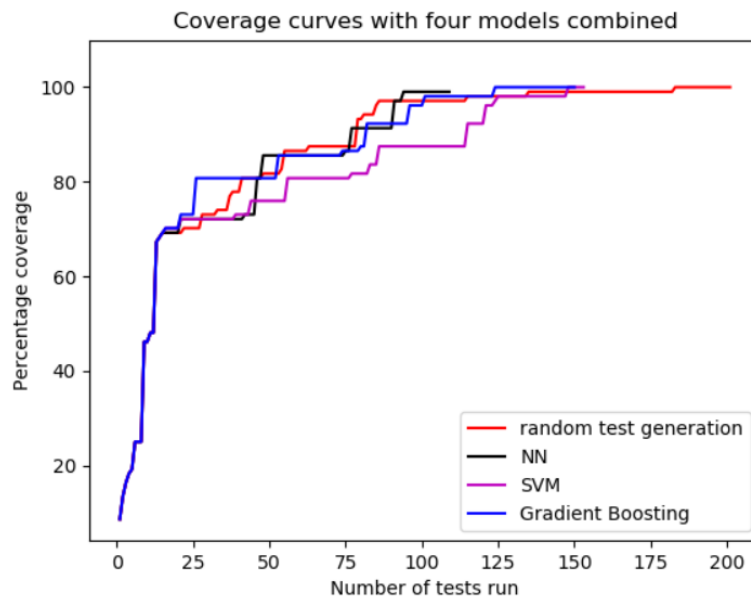


Figure 5-10 Coverage performance curves for each model having combined the model results of four coverpoint, which totally has 122 coverbins.

Table 5-6 Time and test count improvement for all three model construction methods.

Test			
	SVM	GB	NN
Using 280 bins	-7.65027	-1.63934	11.47541
Using 122 bins	6.010929	19.12568	22.95082
Using a combination of individual coverpoints	19.67213	24.59016	44.80874
Time (s)			
	SVM	GB	NN
Using 280 bins	-7.01496	-1.56934	9.930475
Using 122 bins	15.32371	27.13901	30.58514
Using a combination of individual coverpoints	17.88201	29.72002	42.10421

6. CONCLUSION

With the present-day methods used in the industry, there is an increase in time required for coverage closure with increasing design complexity. This work provided a possible solution to this with the help of various machine learning models. We have seen that we were able to achieve coverage closure with fewer number of tests. Following the steps in the three implementations and the reasons for them, we can conclude that neural network provides the best results when compared to SVM and Gradient Boosting algorithms. Also, we can conclude that having a separate model for each coverpoint proves to be better in terms of improvement as well as the data we handle. The choice of coverpoints also matters since they contribute to the number of input features to the models and in turn their accuracy.

REFERENCES

- [1] Stan Sokorac. Optimizing random test constraints using machine learning algorithms. In Design and Verification Conference (DVCon), 2017.
- [2] Fine, S., & Ziv, A. (2003, June). Coverage directed test generation for functional verification using bayesian networks. In Proceedings of the 40th annual Design Automation Conference (pp. 286-291). ACM.
- [3] Chen, W., Wang, L. C., Bhadra, J., & Abadir, M. (2013, May). Simulation knowledge extraction and reuse in constrained random processor verification. In Proceedings of the 50th Annual Design Automation Conference (p. 120). ACM.
- [4] Guzey, O., Wang, L. C., Levitt, J. R., & Foster, H. (2010). Increasing the efficiency of simulation-based functional verification through unsupervised support vector analysis. IEEE transactions on computer-aided design of integrated circuits and systems, 29(1), 138-148.
- [5] Ioannides, C., & Eder, K. I. (2012). Coverage-directed test generation automated by machine learning--a review. ACM Transactions on Design Automation of Electronic Systems (TODAES), 17(1), 7.
- [6] Wang, F., Zhu, H., Popli, P., Xiao, Y., Bodgan, P., & Nazarian, S. (2018, May). Accelerating Coverage Directed Test Generation for Functional Verification: A Neural Network-based Framework. In Proceedings of the 2018 on Great Lakes Symposium on VLSI (pp. 207-212). ACM.

- [7] Denison, D. D., Hansen, M. H., & Holmes, C. (2003). The Boosting Approach to Machine Learning: An Overview. In Nonlinear Estimation and Classification. Springer.
- [8] Mason, L., Baxter, J., Bartlett, P.L. and Frean, M.R., 2000. Boosting algorithms as gradient descent. In Advances in neural information processing systems (pp. 512-518).
- [9] Chollet, Francois and others, Keras,(2015). Retrieved from <https://keras.io>.
- [10] Fabian, Gal and others Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research, 12, 2825-2830 (2011). Retrieved from <https://scikit-learn.org/>.
- [11] Jinde Shubham, Support Vector Machines (SVM) (2018). Retrieved from <https://medium.com/coinmonks/support-vector-machines-svm-b2b433419d73>
- [12] Prince Grover, Gradient Boosting from scratch (2017). Retrieved from <https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d>
- [13] File: Diagrama MESI.GIF. (2017, July 15). Wikimedia Commons, the free media repository. Retrieved from https://commons.wikimedia.org/w/index.php?title=File:Diagrama_MESI.GIF&oldid=251867950.